MAKE THE MOST OF

AWS LAMBDA

WITH

Go

Build, test and deploy great Lambda functions using the Go programming language.

KEVIN MCCONNELL

Table of Contents

Preface

My AWS journey began over a decade ago, when my friends and I first started deploying our applications to EC2 servers, eager to take advantage of this amazing new environment where you could launch new compute capacity in seconds, rent it by the hour, and delete it when you were finished.

Compared to the long leases, up-front purchases, and chilly data center visits we were all used to, this was clearly *the future*, and we were all quite excited about it.

Looking back on it now, though, it's clear that what we were trying to do then was find a better data center: to swap the physical machines we used to use for these new, ephemeral virtual machines that we could order more quickly; to swap some CapEx for OpEx; to buy what we needed now instead of what we guessed we would need next year.

It was an improvement, for sure. It made our jobs a little easier, saved us some money, and it was a lot of fun to boot.

But it turns out that wasn't *the future* after all.

Fast forward a dozen or so years, and it's clear now that the really significant benefit of building for the cloud is in all the ways you can save time and effort by *doing less*. Quickly spinning up a server to run your database on is one thing, but being able to spin up a complete working database—fully maintained and supported by someone else—is quite another.

We now have access to hundreds of managed products and services that take away the burden of installing, patching, troubleshooting, upgrading and maintaining those things ourselves, which leaves us with a lot more time to focus on building our own creations.

It's starting to look a lot more like *that's* the future.

AWS Lambda has a special role in this new world of highly-managed, low-maintenance software products. It is designed to let you run the tiniest piece of your code — a single function — but at the same time it's scalable enough to build whole applications on. It integrates with dozens of other AWS services, letting you glue things together and extend built-in behavior. And it's cheap enough that there are people running their workloads on it for almost nothing.

Used well, Lambda can be an invaluable tool when it comes to building cloud applications. It just takes a little familiarity to know *how* to use it well. My hope is that this book will help you quickly gain that familiarity!

Why I wrote this book

I've spent the last few years building production applications to run on AWS, and during that time I've been leaning more and more on Lambda as a critical component of those applications.

I've been using Go to build these applications, after discovering how much its simplicity and readability helped in writing maintainable code; and how it's speed and memory efficiency was so well suited to the Lambda environment.

Go and Lambda proved to be a great fit, but the details of how to use them effectively wasn't always obvious at first. It took some trial and error to figure that out. After working with these tools for a couple of years, my teammates and I had a much better idea of what worked well for us, and what didn't.

Whenever new engineers joined the team, I would find they often started out with the same questions or difficulties that we'd had, but as we explained our way of using these tools, they quickly became more productive and confident in their use of them.

This book contains what we learned, in the hope that it can save you some of that same trial and error.

Introduction

It's always nice when technologies are not only useful, but fun.

I've found that to be the case with the two technologies that this book is about: AWS Lambda, and the Go programming language.

Both of them can help you to build your projects more easily, to create applications that are fast and scalable, and which are easy to adapt and change as your projects evolve. They're also both small, and nimble tools — quick to learn, but capable of doing a lot.

This "small but mighty" character is at the heart of why they work so well together. And, in my opinion, it's a big part of what makes them fun to use.

Of course, like all tools, there's a knack to using them effectively. And while neither takes long to pick up, there are a handful of techniques that may not be immediately obvious, but which can improve your experience of writing, running, and maintaining your Go-based Lambda applications.

This book will introduce you to these techniques, and guide you, as quickly as possible, to what I hope will be a new level of productivity and enjoyment when building applications.

Why Lambda?

AWS has evolved considerably over the years, and the number of managed services it offers has increased. The more you lean on these managed services when building your applications, the less you have to do yourself. And that saves you time not only when building, but also the operations and maintenance that goes along with it.

Lambda gives you a way to run you own custom code with the operational savings of a managed service. You don't have to think about servers, scaling,

upgrades and patching, and capacity planning (or at least, you don't have to think about them much).

You also don't have to pay for what you're not using. So as great as it is to be able to scale *up* when you need to, it can be equally empowering to launch applications that scale *down* to zero when they aren't being used. If you're building a new product or trying to ship an MVP that doesn't yet have a ton of customers, using Lambda means you may very well be able to run the whole thing for almost nothing (possibly *actually* nothing, if your usage says within the free tier).

Even if you're not building a whole application around Lambda, there are still many places where you'll find it useful.

If you have an existing monolithic application, Lambda functions can be a great way to start splitting out non-core functionality into separate services.

Lambda functions also make great cron jobs. And, paired with an SQS queue, they can be a great way to run background workers.

Lambda is also the main way to "glue together" and customize the other AWS services. If you want to trigger an operation whenever files are uploaded, or write an authentication plugin for an API, Lambda functions are the way to do it.

You can use Lambda to intercept and modify CDN requests and file downloads, or run custom code whenever an EC2 instance is started or stopped in your account. You can even write Lambda functions that add new capabilities to PostgreSQL databases.

Whether you decide to go all-in with Lambda or just want to use a few functions to help out in key areas, I think you'll find it to be a very productive and creative way to build software.

Why Go?

Go turns out to be a great language for writing Lambda functions, for quite a few reasons.

It's high level enough that you can be very productive in it, but still compiles

to very efficient code. Meaning that you can quickly write functions, and still have them be incredibly fast to run.

Go binaries boot up very quickly, which leads to some of the best cold start times you can get.

Go also makes it easy to take advantage of multiple CPUs, which means for certain tasks you can get even more done per millisecond of execution time.

The standard library and build tools are also very high quality and comprehensive, so you have most of the things you need right out of the box.

Of course, there are times when another language will be a better fit for a particular Lambda function. For example, if you're working with machine learning you might find Python to be a better choice given its rich ecosystem of machine learning-related libraries. And one of the nice things about building your application on Lambda is you can mix and match the languages you use for different part. Still, after having written Lambda functions in several different languages over the last few years, I've come to the conclusion that whenever you're free to choose which language to use, you will rarely go wrong by picking Go.

Who this book is for

I wanted this book to be useful to folks who already have some programming experience, who either use Go already or are interested in learning it, and who want to know how to make best use of AWS Lambda in their applications. I've also tried to be concise where possible, to help you learn a lot in a short time.

Because of this, the book doesn't try to teach you Go. There are already lots of great resources for learning Go—many of them freely available—and so covering that in this book would be redundant. I've listed some of my favorites in the appendix. If you're new to the language, I'd recommend trying some of those to see which you like.

Most people find Go to be quick and fun to pick up (relatively speaking, of course), so chances are you'll be up to speed in no time!

The book doesn't assume any prior AWS knowledge. If you've used any part of AWS before, even just to experiment and look around, that's great. But even if you haven't you should be fine.

That said, AWS is a large topic these days, so as you work through the book and learn about the half-dozen or so AWS services that we cover in it, you might also find it helpful to read about, and experiment with, some of the other services as well. The appendix has some resources that you might want to check out for that too.

A lot of the value of Lambda, and of AWS more generally, is in the increasingly wide selection of available services and the way they interact with each other. As you learn more about that landscape you'll start to spot many opportunities to do things more easily.

How to read this book

If you're new to Lambda, or AWS in general, I would suggest reading through the chapters in order. The first few chapters build on each other, to take you through an understanding of what Lambda does (and how it does it), then how to structure your projects and code when using it, before moving on to guidance in how to write your Lambda functions well.

If, however, you prefer to jump straight in to the parts that catch your eye, that's totally fine too. Most of the later chapters are more self-contained, so you can start wherever you like. If you find you get lost with some of the code examples when reading those later chapters, you can always refer back to the earlier chapters for clarification: your best bet will be either the chapter on Code Structure for context of why the Go Lambda code looks the way it does, the Deploying chapter for background on the CDK TypeScript code used to deploy the functions and their infrastructure, or Introducing Lambda for general information about the service.

Chapter 1. Introducing Lambda

1.1. When to use Lambda

There are many ways to run custom applications in AWS.

They vary in terms of their capabilities, pricing structures, and how much of the operational burden they handle for you, like provisioning and maintaining servers.

At one end of the scale you have EC2, which allows you to create virtual servers. You can run any code you like on these servers, for as long as you like, with very few restrictions.

You are, however, responsible for managing the operations of your EC2 servers. If you need more capacity, it's up to you to decide how many servers to run, and which types and sizes they should be. When your capacity requirements fluctuate, you're responsible for adding and removing those servers to match, so you don't end up under-provisioned nor paying for more capacity than you need.

With EC2 you are responsible for patching and upgrading the operating system, installing and maintaining dependencies, setting up monitoring and logging, and so on.

You can integrate with any other services and applications that you like from your EC2 servers, including all the services within AWS; however you will usually have to write the code for those integrations yourself.

Ultimately, EC2 gives you a huge amount of flexibility, at the cost of operational responsibility.

At the other end of the scale is Lambda.

Lambda requires that your code executes within a more restricted environment, but in return, the service handles the vast majority of the operations. Server capacity will be added and removed automatically to handle the load; the underlying operating systems will be patched and maintained for you; and support for logging and monitoring is built in.

With Lambda, you're charged for the capacity that you *use*, rather than what you've provisioned. Your costs will go up and down along with the fluctuations in your workload. You pay less when your application is less busy, and you don't pay anything at all when it's idle.

Lambda also has built-in integrations with many other AWS services, which means you can often use these services without writing any code.

Compared to EC2, Lambda lets you to trade some flexibility in return for convenience.

And in many cases, that turns out to be a *very* good deal!

The operational overhead of running software can often be considerable. Not only do you have the day-to-day aspects of keeping up with security patches and monitoring system health, you also have to consider longer term tasks like capacity planning, system integrations, and periodic platform upgrades. Often this work is not even particularly specific to your applications; it's the same generic keeping-the-lights-on work that every other software company is doing too. The time spent learning and performing this work is time taken away from building your applications.

Lambda can't take *all* of that operational work away, but it can take away the vast majority of it. When you start building your applications on Lambda, and the rich set of managed AWS services that it works with, you will quickly find that a lot of the time you used to spend on operations disappears, and what you're left with often becomes easier.

The idea of *no-ops* might still be a dream, but *hardly-any-ops* is very much a reality.

And while the tradeoff means you'll be running your code in a more restricted environment, the good news is that many applications won't have

any problems operating within those restrictions.

While Lambda was originally more limited in terms of the code you could run (being limited to only a few languages, short running times, and not always the best performance), today Lambda functions are much more capable. Functions that you write now can (among other things):

- access up to 10GB of memory
- be written in any language
- use up to 6 CPU cores at a time
- run for up to 15 minutes per invocation
- connect to a VPC when needed
- use attached network storage

As well as becoming more capable over time, the Lambda service has also seen several significant performance improvements. There is still some overhead involved in running code this way, but the speed at which Lambda can launch processes and perform invocations has increased over time. For many applications the overhead the service introduces won't be a problem.

Given that Lambda functions also have access to the capabilities of the other managed services in the ecosystem, it has turned out to be a great fit for more and more use cases.

With all this in mind, you might be surprised by how much of your application works well in the Lambda environment. Some of the projects I've built and run on Lambda, or helped others build, include:

- Entire back-end APIs for web and mobile applications
- A high-traffic WebSocket-based chat server
- A service for on-demand streaming of short demo videos
- Security and infrastructure management tooling
- Report generation and file export tasks for legacy applications
- An image & video processing pipeline
- Sending and tracking custom emails

Not only did building this way save us time and effort, but in some cases the cost of running the applications was a fraction of what it would have been otherwise.

In fact, whenever I'm starting a new project today, I generally default to building it on Lambda unless I have a good reason not to.

1.2. When not to use Lambda

Despite the strengths of the Lambda platform, there are still some cases where Lambda is not the best choice (or even a viable one) for certain applications.

The most common requirements that can be hurdles to using Lambda are:

Low tail latency

As fast as Lambda is, it does still take it some time to start up function processes. Typically each Lambda process can handle many invocations with very little overhead each time. However, whenever an additional process has to be started (in response to increased load, or just because it's the first time that function has been invoked in a while) there is a longer period while the service provisions a new environment for it. This adds a bit of latency to that invocation. Typically this latency will be on the order of a few hundred milliseconds, in addition to the time it takes your function code to start, but it can be longer.

Depending on your workload, these occasional "cold starts" may or may not matter. For a lot of uses cases the occasional latency spike is fine. But if you have an application where you must serve every request within a short time, then Lambda might not be the best choice for it.

Lambda does offer a potential solution to this problem: *provisioned concurrency*, which is a way to keep a minimum number of process instances ready at all times, reducing (or entirely avoiding) the case where an invocation is forced to wait on a new environment being prepared. If you don't mind paying a bit extra to have this standby capacity, it can be a good solution. However to use it reliably you'll need to be able to predict your capacity needs and provision the concurrency to match, which

means it's not a great fit for every workload.

If you're not sure whether your application can tolerate Lambda's latency, I would encourage you to take measurements and do experiments to see what's actually acceptable in practice. Often a performance metric can seem like a glaring problem to us as engineers, without being particularly noticeable to end users. So it's well worth verifying the real impact before you decide!

Very long running times

If you need to run single tasks for longer than 15 minutes, and there isn't a reasonable way to split them up into smaller pieces, then you won't be able to run those tasks on Lambda (at least not currently).

Lambda works best with shorter, stateless operations, and not every operation fits that category.

If you *do* have that sort of workload but are interested in trying to make it work on Lambda, consider whether there's a way to split your tasks into pieces that can be worked on in parallel. When tasks can be split this way, you'll sometimes find that Lambda's ability to quickly ramp up many thousands of processor cores for the duration of a single task means it can perform the work more quickly (and more cheaply) than other environments.

When cost optimizing stable, high traffic workloads

Lambda is able to be cost effective for many workloads by charging only for the compute resources that are actually used; it is continually scaling up and down to match your traffic, scaling all the way down to zero whenever it can. Although this works great for a lot of use cases, if you have very high traffic at a mostly steady rate, there is not much scope for this scaling to help. At that point, you can often find cheaper ways to serve the requests, such as long-running services on EC2 instances, or using AWS Fargate to run your application in containers.

Moving to such an environment does mean taking on some more of the operational effort that Lambda would have managed for you. So if you're considering doing that, try to determine what that extra work will cost you

in practice, and factor that in as well, rather than only comparing the per-request costs. A strategy that can often work well is to start out using Lambda, and move specific portions of your traffic onto other self-managed platforms later, if and when you find a compelling case for doing so. You may find that the benefits of Lambda turn out to be worth the cost even as you grow, but the great thing about software is that you can always change your mind later if it's not.

From my own experience over the last few years I've found, more often than not, that the applications I was building worked well in the Lambda environment — even in some cases where I initially thought they wouldn't.

1.3. How Lambda works

Although you don't need to know the details of how Lambda works in order to use it, it can help to have some *mechanical sympathy* with it. Knowing what is involved in its various operations can help you make decisions that will lead to better performance and reliability from your functions.

Some of the specific details of the Lambda service will be covered in later chapters (for example, we'll talk about how Lambda does logging in the logging chapter). However in this section we'll go over all the fundamental concepts, to ensure you have a good framework for thinking about how Lambda operates.

At it's heart, Lambda is an API-driven service for defining and executing **functions**. The service responds to API requests to **invoke** these functions, as well as to create and modify their definitions.

While you can (and sometimes will) invoke Lambda functions by using the API directly, it's more common for them to be invoked by one of the many **service integrations** that connect Lambda to the wider AWS environment. These integrations trigger invocations by sending **events** for the function to process.

User-defined code is packaged into a **deployment package**, which can be a simple Zip file or a Docker-style container image. When you create or update a function you include information about where Lambda can access your deployment package, like an S3 bucket or container registry.

The code for a Lambda function has exactly one **handler**, which is the entrypoint used for each invocation. Whenever the Lambda service invokes your function, it calls your handler.

1.4. Process management

Lambda will start as many **instances** of the function as needed to keep up with requests. Whenever it receives a request to invoke a function, it will use an existing idle instance if there is one; if there isn't one ready, it will provision a new one through a process known as a **cold start**.

This cold start process involves fetching the code from the deployment package, and copying it into a new **execution environment**, which is a lightweight virtual machine set up according to the function's configuration. Lambda also performs an **init** phase at this point, which is where the function's code is loaded and started. This is where your code's `main` is run, and it's also where you can do any package-level initialization.

At this point, with the function loaded into the execution environment and initialized, Lambda can actually invoke the function. It calls your code's handler function, passing in whatever data it was told to pass (along with some extra context information of its own). It also captures the return value of your handler, along with any error that occurred, to report back to the caller.

When the handler has finished, Lambda freezes the environment to stop the process running. However it keeps the frozen execution environment around so that it can reuse it to serve subsequent requests more quickly, without having to go through the cold start and init phases every time. (Reusing an environment in this way is known as a **warm start**). If the frozen execution environment isn't used for a while, Lambda will dispose of it to free up resources, which is why a function that is only invoked very occasionally will experience more cold starts.

If you have a function that is invoked regularly and at a steady rate, it typically won't encounter very many cold starts, since the number of running environments will be stable enough that each one will be reused for many invocations (AWS estimates that, on average, about 0.1% of Lambda invocations are cold starts in production applications).

However if your function is only executed rarely, or if its rate of use ramps up and down a lot, cold starts will be more frequent.

In later chapters we'll discuss a few ways of monitoring how often cold starts are happening and how long they are taking, as well as steps you can take to mitigate the impact that they have.

In practice there are more details to Lambda than this (as we'll see throughout the rest of the book) but, in a nutshell, that's what Lambda does: it's a service for running user-defined functions, which it does by transparently managing compute resources on your behalf.

1.5. Runtimes

In order for Lambda to call a handler function, it needs some way of communicating with the function code. It does this through an intermediary layer called a **runtime**.

The runtime makes requests to the Lambda service to get details of the next requested invocation, calls the handler function to perform that invocation, and reports the result back to the service.

Since each language has its own way of calling functions, runtimes are language-specific. There are supplied runtimes for a handful of different supported languages, including one for Go. If you deploy a Lambda function configured to use the `go1.x` runtime, it will run in an environment where this Go runtime is present. The github.com/aws/aws-lambda-go library that you'll use to build Go Lambda handlers knows how to communicate with that runtime, and will do so on your behalf so that your function is invoked correctly.

Rather than use one of the supplied runtimes, you can write your own

custom runtime that communicates with Lambda's runtime API directly. This provides a way to build Lambda functions in any language; not just the ones with AWS-supplied runtimes. Functions that provide their own runtime run in a language-agnostic Linux environment (the runtime names `provided` and `provided.al2` refer to this environment; `provided` runs Amazon Linux, and `provided.al2` runs the newer Amazon Linux 2).

When you build a Go-based Lambda function, the `aws-lambda-go` library also includes the code for a custom runtime in your compiled binary. If you deploy a Go-based Lambda to `provided.al2`, it will automatically use that built-in custom runtime.

Some features are only supported when running on `provided.al2` (we'll get to which features these are later in the book). As a result, the `go1.x` runtime is becoming obsolete at this point. So when deploying your Go-based Lambda functions, you should always configure them to use `provided.al2`. (If you use the CDK to build and deploy them, as we do in this book, it will use `provided.al2` by default, so you don't have to do anything.)

1.6. Configuring functions

Lambda functions have associated configuration that specifies the settings and permissions to use when invoking them, as well as any additional resources that the function needs.

Most functions will only need a subset of the possible configuration options, but there are a few items that every function uses, such as:

- The **function timeout**. If a function invocation doesn't complete within this time, it will be terminated.
- The **memory** allocated to the function. Importantly (and perhaps surprisingly) the amount of CPU power allocated to a function also depends on the memory setting: from a fraction of a CPU core at the least amount of memory (128MB) up to 6 full CPU cores at the largest amount of memory (10GB). Functions with more memory also have better network performance. In effect, the memory setting is more like a turbo boost for your function—you can dial this up to get better performance, or dial it down to save costs.

Interestingly, while functions with more memory cost more per-millisecond to run, they can sometimes work out cheaper in practice because they finish more quickly. It's always worth experimenting with the memory configuration of your functions to find the best balance for your use case.[1]

- The **execution role** used to grant permissions. Whenever a Lambda function runs, it runs with the permissions of its execution role. If you need to grant it the ability to connect to a particular resource, you do so by modifying the role. In this way you can grant each function only the permissions it actually needs, allowing you to easily adopt the principle of *least privilege*.

  At a minimum, Lambda functions use a role that allows them to report their logs to CloudWatch. In practice, most functions will interact with other resources, and so will tend to have additional permissions added to their execution roles to allow that.

- Which **runtime** to use, as discussed in the previous section.

Other very common configuration options include:

- **Environment variables**, which provide a straightforward way to pass configuration to your functions without hardcoding them. If your function accesses other AWS resources, such as S3 buckets or DynamoDB tables, environment variables are a great way to pass in the names and locations of these resources.
- **Concurrency limits** that control how many instances of your function Lambda can run, or that specify a minimum amount that should be kept warm at all times in order to reduce cold starts.
- **Monitoring options** like enhanced metrics and tracing (we'll discuss these in detail in the metrics and tracing chapters).
- A **VPC network** to connect to, allowing your functions to access other resources that are inside a VPC, like relational databases and caches.
- An **EFS filesystem** to attach, if your function needs to read and write to shared network storage.

Typically when writing Lambda functions you will start with a minimal configuration, and add to it as required: adding permissions as needed to its execution role, adding environment variables to pass in data, and so on.

You can also start out with a modest amount of memory for your functions at first, and then experiment with increasing it to see how much the performance improves.

1.7. Pricing

As we've mentioned before, Lambda is billed according to the resources that you use: every time Lambda invokes one of your functions, you are charged something for that invocation.

The more memory you have allocated to a function, the more compute resources it needs, and so functions with more memory are more expensive to run.

Similarly, the longer an invocation takes to finish, the more compute resources it will have used during its run, and so functions that take a long time to run also cost more.

To calculate the cost of each invocation, Lambda multiplies the duration it ran for (rounded up to the nearest millisecond) and the memory allocated, resulting in a number of "GB/seconds" of compute used. A function with 1GB allocated to it that ran for 99.5ms would be counted as 0.1 GB/seconds (0.1s of billable duration, multiplied by the 1GB of memory allocated).

Each AWS region has its own pricing rates for GB/seconds, which are used to calculate the cost.

You're charged for both the invocation time and any init time when your code starts up from a cold start, although you are not charged for the cold start itself. In other words, you pay for the time *your* code is running; you don't pay for any additional time taken by the Lambda service.

There is also a small *per-request* charge for each function invocation.

Lambda's rates are cheap enough that you can generally perform a lot of work without running up a large bill—especially if you can make sure your

functions start and finish quickly (one of the many benefits of using a fast language like Go).

For example, if you have a function with 512MB of memory that takes 10ms to run on average, you can run that function one million times for around 25 cents.

Lambda also has a non-expiring free tier, which includes one million invocations and 400,000 GB/seconds of compute time each month. If you're starting out with a new application, you may find that it takes a while for your usage to grow beyond this free tier, which can make Lambda one of the cheapest ways to launch a new product.

If you do find your usage is increasing steadily over time, you can consider purchasing a Compute Savings Plan, which lets you commit to a minimum level of usage in return for decreased rates.

When performing cost estimates for your projects, bear in mind that the other services you use from your functions have their own pricing models, so you should consider the cost of those services too. If you build a Lambda-based application that stores data in DynamoDB, you'll need to factor in the usage charges from DynamoDB, and so on.

Another thing to bear in mind is that AWS charges for data transfer, and this applies to Lambda in the same way it applies to most of the other AWS services. If you have an application that will use a lot of bandwidth—perhaps you're serving a lot of large media files, for example— then these data transfer costs can be quite significant. So be sure to factor in these costs as well.

You can use the AWS Pricing Calculator (at calculator.aws/) to build detailed cost estimates of your projects, which can be a useful way of comparing the running costs of different architectures before you write the code.

[1] There are some tools that can help automate the process of trial and error in finding the best memory settings. Check out docs.aws.amazon.com/lambda/latest/operatorguide/profile-functions for one popular option.

Chapter 2. Other AWS services to know about

Throughout this book we've mentioned that a big part of what makes Lambda so useful is that it exists within, and works with, such a rich ecosystem of managed services.

For the purposes of this book, you don't need to know any of these other services in great detail. But it will be helpful to have a general familiarity with a few of the common ones, both in terms of what they do, and why they're useful. We'll also refer to some of these services when we look at examples of Lambda code.

Also, as you build more Lambda-based applications, you'll increasingly find uses for these other services. So taking some time to get to know them will definitely pay off.

There are literally *hundreds* of services in AWS these days. But when it comes to building Lambda-based applications, the following are some of the most handy and commonly used.

2.1. API Gateway

A managed API service, API Gateway lets you publish HTTP and WebSocket APIs. The endpoints in these APIs can be implemented by other HTTP services, other AWS services, and—of particular interest to us—Lambda functions. Using Lambda with API Gateway is an easy way to build scalable, serverless APIs, and is one of Lambda's most common use cases.

In addition to publishing the APIs, API Gateway provides a number of useful supporting functions that production APIs often need, including rate limiting, authentication & authorization, and monitoring.

In some cases API Gateway can integrate directly with another AWS service without you needing to write any code to handle the request. For example it

can write data from requests directly into SQS queues and DynamoDB tables.

API Gateway supports two styles of HTTP-based APIs (in addition to WebSocket APIs): REST APIs and (the awkwardly-named) HTTP APIs. REST APIs are the original, and most fully-featured version, but the newer HTTP APIs are cheaper, faster and simpler to use. Unless you need one of the functions that only REST APIs provide, HTTP APIs are usually the best choice for building APIs with Lambda.

2.2. DynamoDB

DynamoDB is a fully-managed, key-value NoSQL database. It's fast and incredibly scalable. And it's often cheaper and easier to run than a relational database.

You can use DynamoDB for your applications' main data store. Or you can use it alongside a relational database to help scale the areas of your application that have the highest activity. It supports a rich variety of data types, and can, with the appropriate modeling, efficiently store and retrieve data with complex relationships.

DynamoDB supports the same *per-use* pricing model as Lambda and API Gateway, and using these three together can be a great way to build back-end APIs for your projects. APIs built around Lambda, API Gateway and DynamoDB can provide almost limitless scaling with predictable performance, yet they cost almost nothing to run while you have more modest traffic.

Despite its benefits, DynamoDB is a lot less flexible than a relational database, and learning to model data effectively with it can also be daunting—especially if you have a lot of prior experience with relational databases. So there are definitely tradeoffs to consider. Even so, it's a very useful item to have in your toolbox.

2.3. S3

Perhaps one of AWS's best-known services, S3 provides object-based file storage. S3 is very often used to store application data and assets. It is also

used by other AWS services—for example, the main way to upload your function code to Lambda is by way of S3.

S3 has a couple of very useful integrations with Lambda. Firstly, it can trigger Lambda functions whenever objects are created or deleted, which can be handy if you need to perform any post-processing of files—for example, to perform feature detection on uploaded images—or as a way to execute downstream business logic whenever files change.

Secondly, the *S3 Object Lambda* integration makes it possible for Lambda functions to intervene when objects are read *out* of S3, either to modify or enhance the file content or even to generate the content entirely on the fly. One of my favorite uses for this integration has been to provide download links for customers that generate up-to-date Zip archives of all their data whenever it's requested.

2.4. SQS

Simple Queue Service (SQS) is a distributed message queue service. It is widely used to route messages between different components in an application.

SQS integrates with a number of other AWS Services, including Lambda. In particular it's straightforward to set up a queue that invokes a Lambda function for every message that is sent into the queue. This can be a great way to add resilience to a part of your application—which is especially useful if you have code that relies on a slow or unreliable external service—as it allows you to easily queue pending operations until they can be successfully processed, and retry any that fail.

As well as basic message delivery, SQS has a few features like dead letter queues, delay queues and batching that make it a flexible way to route messages around your applications.

2.5. SNS

Simple Notification Service (SNS) is a pub/sub notification service that can be used to publish messages to a handful of different subscriber types. The supported subscriber types include email, SMS and mobile push

notifications, SQS queues, and, of course, Lambda functions.

SNS, SQS and Lambda can be used together as building blocks for some common architectural patterns like fan-out and parallelization.

2.6. Step Functions

The AWS Step Functions service provides managed state machine workflows that can be used to orchestrate other tasks.

When used with Lambda functions, Step Functions allows you to break complex logic into multiple steps so that each Lambda function only needs to implement one of these smaller, simpler steps. The service controls the order and flow of execution of these steps according to the state machine definition that you provide. It can retry failed steps, and perform different branches of logic depending on the result of each step.

Step Functions also lets you implement longer workflows that involve waiting on external processes to complete, or even wait for human approval between steps.

2.7. IAM

IAM, AWS's identity and access management platform, is the basis for all of the permission management in AWS. When you need to grant permissions to Lambda functions, you do so via IAM's *roles*.

IAM can be complex in places, but most day-to-day use of it tends to be straightforward. The bulk of what you'll use it for when developing Lambda applications is to specify the permissions that should be granted on specific resources—for example, to permit a particular Lambda function to insert data to a DynamoDB table.

2.8. CloudWatch

AWS provides application and architecture monitoring through the CloudWatch product. CloudWatch provides managed solutions for storing and querying log and metric data, and for publishing dashboards consisting of views of that data. It also allows you to configure alarms and notifications

to alert you about problematic conditions.

Many AWS services automatically send data into CloudWatch so that you can examine and troubleshoot behavior. This includes Lambda, as well as almost all the services we discussed in this chapter.

You also can (and should!) publish your own application data to CloudWatch, so that you'll have that level of visibility into your own code as well. We'll cover the details of how to do that later in the book.

2.9. Services as building blocks

Each of the above services is a well-proven, reliable and appropriate choice for the particular problem it solves. If you need a queue, you won't go far wrong by choosing SQS; and so on for the other services.

All of them have a *pay-per-use* model as well, so you can get started with them without any up-front investment or fixed monthly charges.

However one of the most powerful aspects of combining these services with Lambda is that they can serve as building blocks for your application architecture. Using them can save you time and complexity, and in some cases they are what enables you to implement a solution in Lambda that would otherwise be outside its capabilities.

As a simple example, a Lambda function alone can't be used as a long-running server for an HTTP API. Its execution limit prevents it from running continuously, and the lack of a long-lived IP address for the running environments means you won't have a way to make the Lambda process reachable at a particular domain.

But by adding API Gateway to your architecture, you can make the logic in your Lambda functions accessible through a stable API endpoint. In effect, Lambda becomes a viable way to build APIs because it works with API Gateway.

To give another example, I once worked on an application that needed to send individual real-time notifications to very large numbers of connected clients. Sending notifications to all clients at once from a single Lambda was

too slow to be acceptable—even after optimizing the function, it was simply too many client messages to send from a single process in a reasonable amount of time.

The solution was to split the sending function into two parts, and use SNS to coordinate them. The first part generates batches of recipients, which can be done quickly, and then publishes an SNS message for each batch. The second part responds to each SNS message by sending the notifications to all the clients in that batch.

Splitting the task in this way allows SNS to fan out the work to many thousands of function instances when necessary. The result was that notifications could be sent to all clients several orders of magnitude more quickly than the single function version could manage.

Often the key to successfully designing your applications to run on Lambda involves this sort of thinking. By considering Lambda and other AWS services as the building blocks from which to build architectures, you can reframe large problems as a set of smaller pieces. Each of the smaller pieces becomes simpler to implement and test, and the ability to execute, monitor, and retry these operations individually can help you build in resilience and performance.

This is the end of the free sample.

To read more, visit
https://kevinmcconnell.gumroad.com/l/lambda-go-book